

# Neighbor Table Construction and Update in a Dynamic Peer-to-Peer Network\*

Huaiyu Liu and Simon S. Lam

Dept. of Computer Sciences, Univ. of Texas at Austin, Austin, TX 78712

{huaiyu, lam}@cs.utexas.edu

## Abstract

*In a system proposed by Plaxton, Rajaraman and Richa (PRR), the expected cost of accessing a replicated object was proved to be asymptotically optimal for a static set of nodes and pre-existence of consistent and optimal neighbor tables in nodes [9]. To implement PRR's hypercube routing scheme in a dynamic, distributed environment, such as the Internet, various protocols are needed (for node joining, leaving, table optimization, and failure recovery). In this paper, we first present a conceptual foundation, called C-set trees, for protocol design and reasoning about consistency. We then present the detailed specification of a join protocol. In our protocol, only nodes that are joining need to keep extra state information about the join process. We present a rigorous proof that the join protocol generates consistent neighbor tables for an arbitrary number of concurrent joins. The crux of our proof is based upon induction on a C-set tree. Our join protocol can also be used for building consistent neighbor tables for a set of nodes at network initialization time. Lastly, we present both analytic and simulation results on the communication cost of a join in our protocol.*

## 1. Introduction

The main goal of popular peer-to-peer systems, such as Napster [8], Gnutella [4], and Freenet [3], is object (file) sharing. Objects are stored in user machines and transferred from one machine to another upon requests. In this paper, we view such a system conceptually as a network of *nodes*. Each node, representing a user machine, can send messages to every other node in the system using the Internet.

In these systems, objects are generally replicated, with multiple copies of the same object stored in different nodes. Objects are addressed by location-independent names, with *location-independent routing* used to forward one node's query for an object to some node storing a copy of the object. Four desirable properties of a location-independent

routing infrastructure for these systems, presented by Hil-drum, Kubiawicz, Rao, and Zhao in [5], are the following (slightly rephrased):

- P1 Deterministic Location: If an object exists anywhere in the network, it should be located.
- P2 Routing Locality: If multiple copies of an object exist in the network, a query for the object should be forwarded to a nearby copy. Also, routes should have low stretch.<sup>1</sup>
- P3 Load Balance: The load of storing objects (or object locations) and routing information should be evenly distributed over network nodes.
- P4 Dynamic Membership: The network should adapt to joining and leaving nodes while maintaining the above properties.

Napster employs a centralized directory of object locations and clearly does not satisfy P3. The system is also not fault-tolerant since the directory server is a single point of failure. Gnutella and Freenet were designed to have distributed directory information. However, neither of them satisfies P1.

In two recent research proposals, Chord [12] and CAN [10], the main operation is name resolution, i.e., mapping a name (object ID) to a node that stores a copy of the object (or the location of the object). Each system was designed to satisfy P1, P3, and P4. However, these systems do not satisfy P2 because they are not concerned with forwarding a query directly to a nearby object. Furthermore, while a name can be resolved within a small number of application-level hops,<sup>2</sup> the actual distance of each hop through the Internet, from one node to another, may be very large.

Of interest in this paper is the hypercube routing scheme used in PRR [9], Pastry [11], Tapestry [13], and SPRR [6]. Each node maintains a neighbor table storing pointers (IP addresses) to  $O(\log n)$  nodes in the network. These tables constitute the network's routing infrastructure. With additional distributed directory information, PRR tends to sat-

<sup>1</sup>Stretch is the ratio between the distance traveled by a query to an object to the minimum distance between query origin and the object.

<sup>2</sup>The number is  $O(\log n)$  for Chord and  $O(dn^{1/d})$  for CAN, where  $n$  is the number of nodes in the system and  $d$  is the number of dimensions chosen for CAN.

\*Research sponsored by National Science Foundation grant no. ANI-9977267 and Texas Advanced Research Program grant no. 003658-0439-2001.

isfy each object request with a nearby copy. Given *consistent* (definition in Section 3) and *optimal* (that is, they store nearest neighbors) neighbor tables, PRR guarantees to locate an object if it exists, and the expected cost of accessing a replicated object is asymptotically optimal [9].

To implement the hypercube routing scheme in a dynamic, distributed environment, we need to address the following problems:

1. Given a set of nodes, a join protocol is needed for the nodes to initialize their neighbor tables such that the tables are *consistent*. (In what follows, a “consistent network” means a set of nodes with consistent neighbor tables.)
2. Protocols are needed for nodes to join and leave a consistent network such that the neighbor tables are still consistent after a set of joins and leaves. When a node fails, a recovery protocol is needed to re-establish consistency of neighbor tables.
3. A protocol is needed for nodes to optimize their neighbor tables.

Solving all of these problems is beyond the scope of a single paper. In this paper, we focus on designing a join protocol for the hypercube routing scheme. Given a consistent network, and a set of new nodes joining the network using our protocol, we prove that the join process will terminate and the resulting neighbor tables of both existing and new nodes are consistent (assuming reliable message delivery and no node deletion). In particular, our proof holds for *an arbitrary number of concurrent joins*.

Contributions of this paper are the following:

- We analyze the goal of the join protocol and present a conceptual foundation, called *C-set trees*, for protocol design and reasoning about consistency.
- We design a join protocol for the hypercube routing scheme, and present a detailed protocol specification. In our protocol, only nodes that are still in the join process need to keep extra state information about the join process. The join protocol can also be used for network initialization, where initially the network has only one node. Other nodes then join the network by executing the join protocol.
- We present a rigorous proof that the join protocol produces consistent neighbor tables for an arbitrary number of concurrent joins. The crux of our proof is based upon induction on a *C-set tree*.
- We present both analytic and simulation results on the communication cost of a join in our join protocol.

The join protocol presented in this paper provides a solution to problem 1, and part of the solution to problem 2, discussed above. Moreover, the conceptual foundation presented in this paper can be used for designing protocols for

leaving, failure recovery, and neighbor table optimization.<sup>3</sup> Note that since we are only concerned with consistency in this paper, the assumption of optimal neighbor tables is relaxed when we design our join protocol. Interested readers can refer to [5, 2] for methods of exploiting node proximity and optimizing neighbor tables.

PRR includes algorithms for dynamically maintaining directory information when objects are inserted, deleted, and accessed [9]. However it does not have node join and leave protocols. Pastry, Tapestry, and SPRR all have join and leave protocols. The issue of neighbor table consistency after concurrent joins and leaves was raised but not addressed in SPRR [6]. Pastry uses an optimistic approach to control concurrent node joins and leaves because the authors believe “contention” to be rare [11]. A join protocol was presented for Tapestry with a correctness proof [5]. The join protocol is based upon the use of multicast. The existence of a joining node is announced by a multicast message. Each intermediate node in the multicast tree keeps the joining node in a list (one list per entry updated by a joining node) until it has received acknowledgments from all downstream nodes. This approach has the disadvantage of requiring many existing nodes to store and process extra states as well as send and receive messages on behalf of joining nodes. We take a very different approach in our join protocol design. We put the burden of the join process on joining nodes only.

The balance of this paper is organized as follows: In Section 2, we briefly describe the hypercube routing scheme. In Section 3, our conceptual foundation for protocol design is illustrated. In Section 4, a detailed specification of our join protocol is presented. In Section 5, we present a consistency proof of the join protocol as well as an analysis of the protocol’s communication cost. In Section 6, we discuss how to use the join protocol for network initialization, as well as several protocol enhancements. We conclude in Section 7.

## 2. Background: Hypercube Routing Scheme

Consider a set of nodes and a set of objects. Each node or object has an identifier (ID), which is a fixed-length random binary string. (These IDs are typically generated using a hash function, such as MD5 or SHA-1.) Node and object IDs are drawn from the same ID space which can be thought of as a ring.

To present the hypercube routing scheme, we will follow notation and terminology used for PRR [9]. Each node’s ID is represented by  $d$  digits of base  $b$ . For example, a 32-bit ID can be represented by 8 Hex digits ( $b = 16$ ). Hereafter, we use  $x.ID$  to denote the ID of node  $x$  and use  $x[i]$ ,  $0 \leq i \leq d - 1$ , to denote the  $i$ th digit in  $x.ID$ , with the 0th digit referred to as the *rightmost* digit.

<sup>3</sup>This paper is the first in a series of papers we write to address these problems.

The routing schemes of PRR [9], Pastry [11], Tapestry [13], and SPRR [6] can all be viewed as extensions of the hypercube routing scheme in this paper. For these schemes, a query of an object is routed to a node that matches the object in the largest number of suffix (or prefix) digits.<sup>4</sup> The schemes differ in the technique each uses to resolve the final routing hop when there are multiple nodes that match an object in the largest number of suffix (or prefix) digits. These schemes also differ in how they replicate objects and how they provide fault-tolerant routing.

## 2.1. Neighbor table

The neighbor table of each node consists of  $d$  levels with  $b$  entries at each level. (In what follows, we use *neighbor table* and *table* interchangeably.) The entry  $j$  at level  $i$ ,  $0 \leq j \leq b-1$ ,  $0 \leq i \leq d-1$ , referred to as the  $(i, j)$ -entry, in the table of node  $x$  contains link information to nodes whose IDs share the 0th to  $(i-1)$ th digits (that is, the rightmost  $i$  digits) with  $x.ID$ , and whose  $i$ th digit is  $j$ . These nodes are said to be *neighbors* of  $x$ .<sup>5</sup> If multiple nodes exist with the desired suffix of the  $(i, j)$ -entry, then a subset of these nodes, chosen according to some criterion,<sup>6</sup> may be stored in the entry with the nearest one designated as the *primary* $(i, j)$ -neighbor. Each node also keeps track of its *reverse-neighbors*. Node  $x$  is a *reverse* $(i, j)$ -neighbor of node  $y$  if  $y$  is the *primary* $(i, j)$ -neighbor of  $x$ . Figure 1 shows an example neighbor table, in which only primary-neighbors are shown. (Also, IP addresses of neighbors are omitted.) The number to the right of each entry is the desired suffix for that entry. An empty entry indicates that there does not exist a node in the network whose ID has the desired suffix.

Neighbor table of node 21233 ( $b=4, d=5$ )

∧	01233	10233	0233	31033	033	22303	03	01100	0
11233	11233	21233	1233	03133	133	13113	13	33121	1
21233	21233	∧	2233	21233	233	00123	23	12232	2
∧	31233	03233	3233	∧	333	21233	33	21233	3
level 4	level 3	level 2	level 1	level 0					

Figure 1. An example neighbor table

## 2.2. Routing scheme

When node  $x$  sends a message to node  $y$ , it first forwards the message to  $u_1$ , a primary-neighbor of  $x$  at level-0 that shares the rightmost digit with  $y$ .  $u_1$  then forwards the message to its primary-neighbor at level-1 that shares the

<sup>4</sup>PRR routing uses suffix matching, while the other schemes use prefix matching.

<sup>5</sup>The link information for each neighbor consists of the neighbor's ID and its IP address. For simplicity, we will use "neighbor" or "node" instead of "node's ID and IP address" whenever the meaning is clear from context.

<sup>6</sup>In PRR, for example, nodes with minimum communication costs are chosen. Extra neighbors in an entry are used to facilitate object location [9] or for fault tolerant routing [13].

rightmost two digits with  $y$ . This process continues until the message reaches  $y$ . For example, a message sent from node 21233 to destination node 03231 is first forwarded to the primary(0, 1)-neighbor of 21233, which is 33121 in Figure 1, then to the primary(1, 3)-neighbor of 33121, say, 13331, and so on, until it reaches 03231. In this paper, the primary $(i, x[i])$ -neighbor of  $x$  is chosen to be  $x$  itself. As a result, when  $x$  sends a message to  $y$  following the primary-neighbor pointers, instead of starting at level-0, it starts at level- $k$ , where  $k$  is the length of the longest common suffix of  $x.ID$  and  $y.ID$ .

## 3. Conceptual Foundation

In this section, we analyze the goals and tasks for a join protocol to produce consistent neighbor tables. We first analyze the case of a single join, which is straightforward and presents intuition of the protocol design. Then we discuss multiple joins and present the concept of *C-set trees*.

Since in this paper, we are only concerned with consistency, we relax the assumption of *optimal* neighbor tables in the hypercube routing scheme. In what follows, we use the term *neighbor* to mean *primary neighbor*. Thus, to simplify our presentation, we assume that there is only one neighbor in each table entry. Table 1 presents notation used throughout this paper.

Table 1. Notation

Notation	Definition
$[\ell]$	the set $\{0, \dots, \ell-1\}$ , $\ell$ is a positive integer
$d$	the number of digits in a node's ID
$b$	the base of each digit
$x.table$	the neighbor table of node $x$
$j \cdot \omega$	digit $j$ concatenated with suffix $\omega$
$N_x(i, j)$	the node in the $(i, j)$ -entry of $x.table$ , also referred to as the $(i, j)$ -neighbor of node $x$
$ \omega $	the number of digits in suffix $\omega$
$csuf(\omega_1, \omega_2)$	the longest common suffix of $\omega_1$ and $\omega_2$
$\langle V, \mathcal{N}(V) \rangle$	a hypercube network: $V$ is the set of nodes in the network, $\mathcal{N}(V)$ is the set of neighbor tables
$V_{i_1 \dots i_0}$	a <i>suffix set</i> of $V$ , which is the set of nodes in $V$ , each of which has an ID with the suffix $i_1 \dots i_0$
$ V $	the number of nodes in set $V$

### 3.1. Definitions and assumptions

**Definition 3.1** Let  $t_x^b$  be the time when node  $x$  begins joining a network, and  $t_x^e$  be the time when  $x$  becomes an  $S$ -node (to be defined in Section 4). The period from  $t_x^b$  to  $t_x^e$ , denoted by  $[t_x^b, t_x^e]$ , is the **joining period** of  $x$ .

**Definition 3.2** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 2$ , join a network. If the joining period of each node does not overlap with that of any other, then the joins are **sequential**.

**Definition 3.3** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 2$ , join a network. Let  $t^b = \min(t_{x_1}^b, \dots, t_{x_m}^b)$  and  $t^e = \max(t_{x_1}^e, \dots, t_{x_m}^e)$ . If for each node  $x$ ,  $x \in W$ , there exists a node  $y$ ,  $y \in W$  and  $y \neq x$ , such that their joining

periods overlap, and there does not exist a sub-interval of  $[t^b, t^e]$  that does not overlap with the joining period of any node in  $W$ , then the joins are **concurrent**.

**Definition 3.4** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 1$ , join a network  $\langle V, \mathcal{N}(V) \rangle$ . For any node  $x$ ,  $x \in W$ , if  $V_{x[k-1] \dots x[0]} \neq \emptyset$  and  $V_{x[k] \dots x[0]} = \emptyset$ ,  $1 \leq k \leq d-1$ , then  $V_{x[k-1] \dots x[0]}$  is the **notification set** of  $x$  regarding  $V$ , denoted by  $V_x^{Notify}$ . If  $V_{x[0]} = \emptyset$ , then  $V_x^{Notify}$  is  $V$ .

**Definition 3.5** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 2$ , join a network  $\langle V, \mathcal{N}(V) \rangle$ . The joins are **independent** if for any pair of nodes  $x$  and  $y$ ,  $x \in W$ ,  $y \in W$ ,  $x \neq y$ ,  $V_x^{Notify} \cap V_y^{Notify} = \emptyset$ .

**Definition 3.6** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 2$ , join a network  $\langle V, \mathcal{N}(V) \rangle$ . The joins are **dependent** if for any pair of nodes  $x$  and  $y$ ,  $x \in W$ ,  $y \in W$ ,  $x \neq y$ , one of the following is true:

- $V_x^{Notify} \cap V_y^{Notify} \neq \emptyset$ .
- $\exists u, u \in W$ ,  $u \neq x \wedge u \neq y$ , such that  $V_x^{Notify} \subset V_u^{Notify}$  and  $V_y^{Notify} \subset V_u^{Notify}$ .

**Definition 3.7** Consider two nodes,  $x$  and  $y$ , in network  $\langle V, \mathcal{N}(V) \rangle$ . If there exists a neighbor sequence  $(u_0, \dots, u_k)$ ,  $k \leq d$ , such that  $u_0$  is  $x$ ,  $u_k$  is  $y$ , and  $u_{i+1}$  is  $N_{u_i}(i, y[i])$ ,  $i \in [k]$ , then  $y$  is **reachable** from  $x$  (within  $k$  hops), or  $x$  can **reach**  $y$ , to be denoted by  $\langle x \rightarrow y \rangle_k$ .

**Definition 3.8** Consider a network  $\langle V, \mathcal{N}(V) \rangle$ . The network, or  $\mathcal{N}(V)$ , is **consistent** if for any node  $x$ ,  $x \in V$ , each entry in its table satisfies the following conditions:

- (a) If  $V_{j \cdot x[i-1] \dots x[0]} \neq \emptyset$ , then  $N_x(i, j) = y$ ,  $y \in V_{j \cdot x[i-1] \dots x[0]}$ ,  $1 \leq i \leq d-1$ ,  $j \in [b]$ ; if  $V_j \neq \emptyset$ , then  $N_x(0, j) = y$ ,  $y \in V_j$ ,  $j \in [b]$ .
- (b) If  $V_{j \cdot x[i-1] \dots x[0]} = \emptyset$ , then  $N_x(i, j) = \text{null}$ ,  $1 \leq i \leq d-1$ ,  $j \in [b]$ ; if  $V_j = \emptyset$ , then  $N_x(0, j) = \text{null}$ ,  $j \in [b]$ .

If condition (a) is satisfied, then  $\mathcal{N}(V)$  is **false negative free**, i.e., if a node exists in the network, it is reachable from any other node. If condition (b) is satisfied, then  $\mathcal{N}(V)$  is **false positive free**, i.e., if a node does not exist in the network, there should not exist a path that leads to it.

**Lemma 3.1** In a network  $\langle V, \mathcal{N}(V) \rangle$ , any node is reachable from any other node iff condition (a) of Definition 3.8 is satisfied by the network.

In designing our protocol for nodes to join a network  $\langle V, \mathcal{N}(V) \rangle$ , we assume that (i)  $V \neq \emptyset$  and  $\mathcal{N}(V)$  is consistent, (ii) each joining node, by some means, knows a node in  $V$  initially, (iii) messages between nodes are delivered reliably, and (iv) there is no node deletion (leave or failure) during the joining period of any node.

Under the assumption that there is no node deletion during joins, condition (b) in Definition 3.8 can be satisfied easily, since once a node has joined, it always exists in the network. Hence, the goal of the join protocol is to construct neighbor tables for new nodes and update tables of existing

nodes such that condition (a) in Definition 3.8 is satisfied. In a distributed peer-to-peer network, global knowledge is difficult (if not impossible) to get. Therefore, a node should utilize local information to construct or update neighbor tables. Our join protocol is designed to expand the network monotonically and preserve reachability of existing nodes so that once a set of nodes can reach each other, they always can thereafter. Hence, starting with a consistent network,  $\langle V, \mathcal{N}(V) \rangle$ , and a set  $W$  of joining nodes, the goals of the protocol are the following:

- **Goal 1:**  $\forall x, \forall y, x \in W, y \in V$ , eventually  $x$  and  $y$  can reach each other.
- **Goal 2:**  $\forall x_1, \forall x_2, x_1 \in W, x_2 \in W$ , eventually  $x_1$  and  $x_2$  can reach each other.

### 3.2. Operations of a single join

When  $x$  joins, it is given a node  $g_0$ ,  $g_0 \in V$ . First,  $x$  constructs its own table level by level by copying neighbors from nodes in  $V$ . It starts with copying level-0 neighbors of  $g_0$  into level-0 of its own table. Among these level-0 neighbors,  $x$  finds node  $g_1$ ,  $g_1[0] = x[0]$ . Then  $x$  copies level-1 neighbors of  $g_1$  into level-1 of its own table and searches for a node  $g_2$  that shares the rightmost 2 digits with it. This process is repeated until  $x$  cannot find a node that shares the rightmost  $k+1$  digits with it ( $k$  must exist and is at most  $d-1$  since  $x.ID$  is unique in the network).  $x$  then adds itself into its table.

Since there exist nodes in  $V$  that share the rightmost  $k$  digits with  $x$  but no node shares the rightmost  $k+1$  digits with  $x$ ,  $V_{x[k-1] \dots x[0]} \neq \emptyset$ , however,  $V_{x[k] \dots x[0]} = \emptyset$ . Hence nodes in  $V_{x[k-1] \dots x[0]}$  need to be notified and their  $(k, x[k])$ -entries need to be updated. Conceptually, nodes in  $V_{x[k-1] \dots x[0]}$  form a forest whose roots are the level- $k$  neighbors of  $x$ . By following neighbor pointers,  $x$  traverses the forest and notifies all nodes in  $V_{x[k-1] \dots x[0]}$  eventually.

During  $x$ 's join, the consistency of the original network  $\langle V, \mathcal{N}(V) \rangle$  is preserved because nodes in  $V$  will fill  $x$  into a table entry only if that entry is empty.

### 3.3. Operations of multiple joins

If multiple nodes join a network sequentially, then the joins do not interfere with each other, because when a node joins, any node that joined earlier has already been integrated into the network. Also, if multiple nodes join a network concurrently and the joins are independent, then intuitively the joins do not interfere with each other, because the sets of nodes these joining nodes need to notify do not intersect and none of the joining nodes needs to store any other joining node in its table. The most difficult case is *concurrent and dependent joins*, where the views different joining nodes have about the current network may conflict. For example, if nodes 10261 and 00261 join concurrently, each of them may think of itself as the only node with suffix 261

in the network. If handled incorrectly, views of the joining nodes may not converge eventually, which would result in inconsistent neighbor tables.

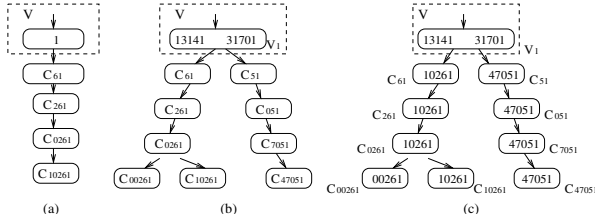


Figure 2. C-set tree

We first analyze the desirable results of multiple joins by using an example ( $b = 8, d = 5$ ). Suppose a set of nodes,  $W = \{10261, 47051, 00261\}$ , join a consistent network  $\langle V, \mathcal{N}(V) \rangle$ ,  $V = \{72430, 10353, 62332, 13141, 31701\}$ . Then, at the end of joins, for any  $y$  to reach 10261,  $y \in V$ , there should exist a neighbor sequence  $(u_0, u_1, \dots, u_5)$  such that  $u_0$  is  $y$ ,  $u_5$  is 10261, and the IDs of  $u_1$  to  $u_4$  have suffix 1, 61, 261, and 0261, respectively. Since  $\mathcal{N}(V)$  is consistent,  $y$  must have stored a neighbor with suffix 1, which can be any node in  $V_1$ . Let the set of (1, 6)-neighbors of nodes in  $V_1$  be  $C_{61}$ , the set of (2, 2)-neighbors of nodes in  $C_{61}$  be  $C_{261}$  and so on. We call these sets *C-sets* and the sequence of sets from  $V_1$  to  $C_{10261}$  a *C-set path*. As shown in Figure 2(a), for nodes in  $V$  to reach 10261, each C-set in the path should be filled with nodes in  $W$  with the desired suffix. Generally, from any node in  $V$  to each node in  $W$ , there is an associated C-set path, and all the paths form a tree rooted at  $V_1$ , called a *C-set tree*, as shown in Figure 2(c). Note that C-set trees are conceptual structures used for protocol design and reasoning about consistency. They are *not implemented* in any node.

The above example is a special case of multiple joins, where the notification sets regarding  $V$  (*noti-sets*, in short) of all nodes in  $W$  are the same (namely,  $V_1$  in the example). Generally, the noti-sets of all nodes in  $W$  may not be the same. Then, nodes with the same noti-set belong to the same C-set tree and the C-set trees for all nodes in  $W$  form a forest. In the above example, if  $W = \{10261, 00261, 67320, 11445\}$ , then 10261 and 00261 belong to a C-set tree rooted at  $V_1$ , 67320 belongs to a C-set tree rooted at  $V_0$  and 11445 belongs to a C-set tree rooted at  $V$ . Each C-set tree can be treated separately. Hence, in the balance of this subsection, our discussion is focused on a single C-set tree.

**Definition 3.9** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 1$ , join a consistent network  $\langle V, \mathcal{N}(V) \rangle$ , and for any node  $x$ ,  $x \in W$ ,  $V_x^{Notify} = V_\omega$ , where  $|\omega| = k$ . Then the **C-set tree template** associated with  $V$  and  $W$ , denoted by  $C(V, W)$ , is defined as follows:

- $V_\omega$  is the root of the tree (the root is not a C-set);
- If  $W_{l_1 \dots l_k} \neq \emptyset$ ,  $l_1 \in [b]$ , then set  $C_{l_1 \dots l_k}$  is a child of  $V_\omega$ , and  $l_1 \dots l_k$  is the associated suffix of  $C_{l_1 \dots l_k}$ ;

- If  $W_{l_1 \dots l_k} \neq \emptyset$ ,  $2 \leq j \leq d - k$ ,  $l_1, \dots, l_j \in [b]$ , then set  $C_{l_1 \dots l_j \dots l_k}$  is a child of set  $C_{l_1 \dots l_{j-1} \dots l_k}$ .

Given  $V$  and  $W$ , the tree template is determined. Figure 2(b) shows the tree template for the above example. The task of the join protocol is to construct and update neighbor tables such that paths are established between nodes; *conceptually* nodes are filled into each C-set in  $C(V, W)$ . For different sequences of protocol message exchange, different nodes could be filled into each C-set, which would result in different realizations of the tree template. Figure 2(c) shows one realization of the tree template in Figure 2(b). Observe that since for any node  $x$ ,  $N_x(i, x[i]) = x$ ,  $i \in [b]$ , once  $x$  is filled into a C-set, it is automatically filled into those descendants of the C-set in the tree, whose suffix is also a suffix of  $x.ID$ . For instance, if both 13141 and 31701 store 10261 in (1, 6)-entry, then conceptually 10261 is filled in  $C_{61}$  and consequently,  $10261 \in C_{261}, C_{0261}$  and  $C_{10261}$ .

Given a set  $W$  of nodes joining a consistent network  $\langle V, \mathcal{N}(V) \rangle$  and nodes in  $W$  belong to the same C-set tree, we denote the C-set tree realized at the end of all joins as  $cset(V, W)$  (formal definition of  $cset(V, W)$  is in Section 5.1). By the end of joins, the following conditions should be satisfied by neighbor tables of nodes in  $V \cup W$  for them to be consistent:

- (1)  $cset(V, W)$  has the same structure with  $C(V, W)$  and none of the C-sets in  $cset(V, W)$  is empty.
- (2) For each node  $y$ ,  $y \in V_\omega$  (root of the C-set tree), for each child C-set of  $V_\omega$  in  $cset(V, W)$ ,  $y$  stores a node with the suffix of that C-set into its neighbor table.
- (3) For each node  $x$ ,  $x \in W$ , the C-set whose suffix is  $x.ID$  is a leaf node in the tree. For any C-set along the path from this leaf node to the root, if it has any sibling C-set, then  $x$  stores a node with the suffix of that sibling C-set in its table.<sup>7</sup>

If condition (1) is satisfied, then in  $cset(V, W)$ , each leaf node whose suffix corresponds to a node's ID must include that node. Therefore, the union of all C-sets in  $cset(V, W)$  is  $W$ . If condition (2) is satisfied, then all entries in  $\mathcal{N}(V)$  that need to be updated are updated. If condition (3) is satisfied, then for any node in  $W$ , it can reach every other node in  $W$ . Hence, these three conditions, together with each joining node's copying neighbors from nodes in  $V$ , ensure that the network is consistent after the joins.

## 4. Specification of Join Protocol

In our protocol, each node keeps its own status, which could be *copying*, *waiting*, *notifying*, and *in\_system*. When a node starts joining, its status is set to *copying*. Each node also stores the state of each neighbor as  $T$  or  $S$  in its table, where

<sup>7</sup>For instance, in Figure 2(c), from  $C_{00261}$  to  $V_1$ , there are two branches with suffix 10261 and 51 respectively. Then, eventually, 00261 should store a node with suffix 10261 and a node with suffix 51.

$S$  indicates that the neighbor is in status  $in\_system$ , while  $T$  means it is not yet.

A node with status  $in\_system$  is called an  $S$ -node; otherwise, it is a  $T$ -node. Figure 3 describes the state variables of a joining node. Variables in the first part are also used by nodes in  $V$ , where for each node  $y$ ,  $y \in V$ ,  $y.status = in\_system$ ,  $y.table$  is populated in a way that satisfies the conditions in Definition 3.8, and  $N_y(i, j).state = S$  if  $N_y(i, j) \neq null$  for all  $i$  and  $j$ . Figure 4 presents the protocol messages. Figures 5 to 14 present the pseudo-code of the protocol, in which  $x$ ,  $y$ ,  $u$  and  $v$  denote nodes, and  $i$ ,  $j$  and  $k$  denote integers. Note that when any node,  $x$ , sets  $N_x(i, j) = y$ ,  $y \neq x$ ,  $x$  needs to send a  $RvNghNotiMsg(y, N_x(i, j).state)$  to  $y$ , and  $y$  should reply to  $x$  if  $N_x(i, j).state$  is not consistent with  $y.status$ . For clarity of presentation, we have omitted the sending and reception of these messages in the pseudo-code.

#### State variables of a joining node $x$ :

$x.status \in \{copying, waiting, notifying, in\_system\}$ , initially *copying*.  
 $N_x(i, j)$ : the  $(i, j)$ -neighbor of  $x$ , initially *null*.  
 $N_x(i, j).state \in \{T, S\}$ .  
 $R_x(i, j)$ : the set of reverse  $(i, j)$ -neighbors of  $x$ , initially *empty*.  
  
 $x.noti\_level$ : an integer, initially 0.  
 $Q_r$ : a set of nodes from which  $x$  waits for replies, initially *empty*.  
 $Q_n$ : a set of nodes  $x$  has sent notifications to, initially *empty*.  
 $Q_j$ : a set of nodes that have sent  $x$  a *JoinWaitMsg*, initially *empty*.  
 $Q_{sr}, Q_{sn}$ : a set of nodes, initially *empty*.

Figure 3. State variables

#### Messages exchanged by nodes:

*CpRstMsg*, sent by  $x$  to request a copy of receiver's neighbor table.  
*CpRlyMsg*( $x.table$ ), sent by  $x$  in response to a *CpRstMsg*.  
*JoinWaitMsg*, sent by  $x$  to notify receiver of the existence of  $x$ , when  $x.status$  is *waiting*.  
*JoinWaitRlyMsg*( $r, u, x.table$ ), sent by  $x$  in response to a *JoinWaitMsg*,  $r \in \{negative, positive\}$ ,  $u$ : a node.  
*JoinNotiMsg*( $x.table$ ), sent by  $x$  to notify receiver of the existence of  $x$ , when  $x.status$  is *notifying*.  
*JoinNotiRlyMsg*( $r, x.table, f$ ), sent by  $x$  in response to a *JoinNotiMsg*,  $r \in \{negative, positive\}$ ,  $f \in \{true, false\}$ .  
*InSysNotiMsg*, sent by  $x$  when  $x.status$  changes to *in\\_system*.  
*SpeNotiMsg*( $x, y$ ), sent or forwarded by a node to inform receiver of the existence of  $y$ , where  $x$  is the initial sender.  
*SpeNotiRlyMsg*( $x, y$ ), response to a *SpeNotiMsg*.  
*RvNghNotiMsg*( $y, s$ ), sent by  $x$  to notify  $y$  that  $x$  is a reverse neighbor of  $y$ ,  $s \in \{T, S\}$ .  
*RvNghNotiRlyMsg*( $s$ ), sent by  $x$  in response to a *RvNghNotiMsg*,  $s = S$  if  $x.status$  is *in\\_system*; otherwise  $s = T$ .

Figure 4. Protocol messages

### 4.1. Action in status copying

In this status,  $x$  constructs its table level by level until it stops at level- $k$ ,  $k \in [b]$ , where after copying level- $k$  neighbors of node  $g_k$ ,  $x$  could not find a node  $g_{k+1}$  that shares the rightmost  $k + 1$  digits with it, or  $x$  finds such a  $g_{k+1}$

but  $g_{k+1}$  is still a T-node. In the former case,  $x$  sends a *JoinWaitMsg* to  $g_k$ , while in the latter case,  $x$  sends a *JoinWaitMsg* to  $g_{k+1}$ . Meanwhile,  $x$  sets its status to *waiting*. Figure 5 depicts the action in this status. (For clarity of presentation, we have omitted the sending of a *CpRstMsg* from  $x$  to  $g$ , and the reception of a *CpRlyMsg* from  $g$  to  $x$ .)

#### Action of $x$ on joining $\langle V, \mathcal{N}(V) \rangle$ , given node $g_0$ , $g_0 \in V$ :

$i$ : initially 0.  $p, g$ : a node, initially  $g_0$ .  $s \in \{T, S\}$ , initially  $S$ .  
  
 $x.status = copying$ ;  
**while** ( $g \neq null$  and  $s == S$ ) { // copy level- $i$  neighbors of  $g$   
  **for** ( $j = 0; j < b; j++$ ) {  
     $N_x(i, j) = N_g(i, j); N_x(i, j).state = N_g(i, j).state$ ;  
  }  
   $p = g; g = N_p(i, x[i]); s = N_p(i, x[i]).state; i++$ ;  
}  
**for** ( $i = 0; i < d; i++$ ) {  $N_x(i, x[i]) = x; N_x(i, x[i]).state = T$ ;  
 $x.status = waiting$ ;  
**if** ( $g == null$ ) {  
  Send *JoinWaitMsg* to  $p$ ;  $Q_n = Q_n \cup \{p\}$ ;  $Q_r = Q_r \cup \{p\}$ ;  
**else** {Send *JoinWaitMsg* to  $g$ ;  $Q_n = Q_n \cup \{g\}$ ;  $Q_r = Q_r \cup \{g\}$ ;  
}

Figure 5. Action in status copying

### 4.2. Action in status waiting

The *JoinWaitMsg*  $x$  sends to  $g_k$  (or  $g_{k+1}$ ) notifies  $g_k$  (or  $g_{k+1}$ ) that  $x$  is waiting to be stored in its table. If  $g_k$  (or  $g_{k+1}$ ) has already stored  $u_1$  in the entry  $x$  can be filled into by the time it receives the message, it sends a negative reply to  $x$  with  $u_1$  and its own table.  $x$  then sends another *JoinWaitMsg* to  $u_1$ . This process may be repeated (for at most  $d$  times) until some node fills  $x$  into its table and sends  $x$  a positive reply. Note that a node can only reply to  $x$  when it is an S-node; otherwise, it has to delay its reply. On receiving a positive reply,  $x$  changes status to *notifying* and sets  $x.noti\_level = |csuf(x.ID, y.ID)|$ , where  $y$  is the node that sends the positive reply to  $x$ . Figures 6 and 7 present actions upon receiving *JoinWaitMsg* and *JoinWaitRlyMsg*, respectively.

#### Action of $y$ on receiving *JoinWaitMsg* from $x$ :

$k = |csuf(x.ID, y.ID)|$ ;  
**if** ( $y.status == in\_system$ ) {  
  **if** ( $N_y(k, x[k]) \neq null \wedge N_y(k, x[k]) \neq x$ ) {  
    Send *JoinWaitRlyMsg*(*negative*,  $N_y(k, x[k]), y.table$ ) to  $x$ ;  
  }  
  **else** { // it must be that  $N_y(k, x[k])$  is *null*  
     $N_y(k, x[k]) = x; N_y(k, x[k]).state = T$ ;  
    Send *JoinWaitRlyMsg*(*positive*,  $x, y.table$ ) to  $x$ ;  
  }  
**else**  $Q_j = Q_j \cup \{x\}$ ;

Figure 6. Action on receiving *JoinWaitMsg*

### 4.3. Action in status notifying

As shown in Figure 8, in this status, if  $x$  finds a node  $y$  such that  $|csuf(x.ID, y.ID)| \geq x.noti\_level$ ,  $x$  sends a *JoinNotiMsg*, which includes  $x.table$ , to  $y$  if it has not done so.

Action of  $x$  on receiving  $JoinWaitRlyMsg(r, u, y.table)$  from  $y$ :

```

 $Q_r = Q_r - \{y\}; k = |csuf(x.ID, y.ID)|;$ 
if ( $N_x(k, y[k]) == y$ )  $N_x(k, y[k]).state = S;$ 
if ( $r == positive$ ) {
   $x.status = notifying;$   $x.noti\_level = k;$ 
   $R_x(k, x[k]) = R_x(k, x[k]) \cup \{y\};$ 
}else { Send  $JoinWaitMsg$  to  $u;$   $Q_n = Q_n \cup \{u\}; Q_r = Q_r \cup \{u\};$ 
  Check_Ngh_Table( $y.table$ );
if ( $x.status == notifying \wedge Q_r == \emptyset \wedge Q_{sr} == \emptyset$ )
  Switch_To_S_Node();

```

Figure 7. Action on receiving JoinWaitRlyMsg

On receiving the  $JoinNotiMsg$  from  $x$ ,  $y$  fills  $x$  into its table if the corresponding entry is empty and replies with  $y.table$ .  $x$  then checks  $y.table$  level by level to send more  $JoinNotiMsg$  if necessary. Also, if  $x$  finds a node  $u$  in  $y.table$  that can be filled into an empty entry, it stores  $u$  in that entry. Figures 9 and 10 present the actions on receiving  $JoinNotiMsg$  and  $JoinNotiRlyMsg$ , respectively.

Check\_Ngh\_Table( $y.table$ ) at  $x$ :

```

for each  $N_y(i, j)$  {
  if ( $N_y(i, j) \neq null \wedge N_y(i, j) \neq x$ ) {
     $u = N_y(i, j); k = |csuf(x.ID, u.ID)|;$ 
    if ( $N_x(k, u[k]) == null$ ) {
       $N_x(k, u[k]) = u;$   $N_x(k, u[k]).state = N_y(i, j).state;$ 
    }
    if ( $x.status == notifying \wedge k \geq x.noti\_level \wedge u \notin Q_n$ ) {
      Send  $JoinNotiMsg(x.table)$  to  $u;$ 
       $Q_n = Q_n \cup \{u\}; Q_r = Q_r \cup \{u\};$ 
    }
  }
}

```

Figure 8. Subroutine: Check\_Ngh\_Table

Action of  $y$  on receiving  $JoinNotiMsg(x.table)$  from  $x$ :

```

 $k = |csuf(x.ID, y.ID)|; f = false;$ 
if ( $N_y(k, x[k]) == null$ ) {  $N_y(k, x[k]) = x;$   $N_y(k, x[k]).state = T;$  }
if ( $N_x(k, y[k]) \neq y \wedge y.status == in\_system$ )  $f = true;$ 
if ( $N_y(k, x[k]) == x$ ) Send  $JoinNotiRlyMsg(positive, y.table, f)$  to  $x$ ;
else Send  $JoinNotiRlyMsg(negative, y.table, f)$  to  $x$ ;
Check_Ngh_Table( $x.table$ );

```

Figure 9. Action on receiving JoinNotiMsg

In what follows, we use “notification” to refer to either a  $JoinWaitMsg$  or a  $JoinNotiMsg$ . So far, three cases for a node  $x$  to know another node  $y$  have been presented: (i)  $x$  copies  $y$  in status *copying*, (ii)  $x$  receives a notification from  $y$ , and (iii)  $x$  receives a message, which includes  $z.table$ , from  $z$  and  $y$  is in  $z.table$ . There is one more case, as shown in Figures 9 and 10: Suppose in status *notifying*,  $x$  sends a  $JoinNotiMsg$  to  $y$ . When  $y$  receives the message, if  $y$  is an S-node and finds that  $N_x(k, y[k]) = u_1$ , where  $k = |csuf(x.ID, y.ID)|$  and  $u_1 \neq y$ , then  $y$  sets a flag in its reply. Seeing the flag in the reply,  $x$  sends a  $SpeNotiMsg$  to  $u_1$  to inform it about  $y$  if  $x$  has not done so and

Action of  $x$  on receiving  $JoinNotiRlyMsg(r, y.table, f)$  from  $y$ :

```

 $Q_r = Q_r - \{y\}; k = |csuf(x.ID, y.ID)|;$ 
if ( $r == positive$ )  $R_x(k, x[k]) = R_x(k, x[k]) \cup \{y\};$ 
if ( $f == true \wedge k > x.noti\_level \wedge y \notin Q_{sn}$ ) {
  Send  $SpeNotiMsg(x, y)$  to  $N_x(k, y[k]);$ 
   $Q_{sn} = Q_{sn} \cup \{y\}; Q_{sr} = Q_{sr} \cup \{y\};$ 
}
Check_Ngh_Table( $y.table$ );
if ( $Q_r == \emptyset \wedge Q_{sr} == \emptyset$ ) Switch_To_S_Node();

```

Figure 10. Action on receiving JoinNotiRlyMsg

$k > x.noti\_level$ . If  $u_1$  has set  $u_2$  instead of  $y$  as the corresponding neighbor, it forwards the message to  $u_2$ . This process stops when a receiver stores or has stored  $y$  in its table and sends a reply to  $x$ . (The process can be repeated at most  $d$  times.) Figures 11 and 12 depict the actions on receiving  $SpeNotiMsg$  and  $SpeNotiRlyMsg$ , respectively.

Action of  $u$  on receiving  $SpeNotiMsg(x, y)$  from  $v$ :

```

 $k = |csuf(y.ID, u.ID)|;$ 
if ( $N_u(k, y[k]) == null$ ) {  $N_u(k, y[k]) = y;$   $N_u(k, y[k]).state = S;$  }
if ( $N_u(k, y[k]) \neq y$ ) Send  $SpeNotiMsg(x, y)$  to  $N_u(k, y[k]);$ 
else Send  $SpeNotiRlyMsg(x, y)$  to  $x$ ;

```

Figure 11. Action on receiving SpeNotiMsg

Action of  $x$  on receiving  $SpeNotiRlyMsg(x, y)$  from  $u$ :

```

 $Q_{sr} = Q_{sr} - \{y\};$  if ( $Q_r == \emptyset \wedge Q_{sr} == \emptyset$ ) Switch_To_S_Node();

```

Figure 12. Action on receiving SpeNotiRlyMsg

#### 4.4. Action in status *in\_system*

When  $x$  has received replies from all of the nodes it has notified and finds no more node to notify, it changes status to *in\_system*. Next,  $x$  informs all of its reverse-neighbors and nodes in  $Q_j$ , which have sent it a  $JoinWaitMsg$ , that it has become an S-node. Figures 13 and 14 present the pseudo-code for this part.

## 5. Protocol Analysis

In this section, we present a consistency proof of the join protocol, and analyze the communication cost of each join. Due to space limitation, we only present important lemmas and proof outlines. Proof details can be found in [7].

### 5.1. Correctness of join protocol

We present two theorems. Theorem 1 states that when a set of nodes use the join protocol to join a consistent network, then at the end of the joins, the resulting network is also consistent. Theorem 2 states that each joining node eventually becomes an S-node. We begin by presenting Lemmas 5.1 to 5.4. Recall that  $t_x^e$  denotes the time when a joining node  $x$  becomes an S-node. In what follows, we use  $t^e$  to denote  $\max(t_{x_1}^e, \dots, t_{x_m}^e)$ .

```

Switch_To_S_Node() at x:
x.status = in_system;
for (i = 0; i < d; i++) { N_x(i, x[i]).state = S; }
for each v of x's reverse neighbors, Send InSysNotiMsg to v;
for each u, u ∈ Q_j {
k = |csuf(x.ID, u.ID)|;
if (N_x(k, u[k]) == null) {
N_x(k, u[k]) = u; N_x(k, u[k]).state = T;
Send JoinWaitRlyMsg(positive, u, x.table) to u;
} else Send JoinWaitRlyMsg(negative, N_x(k, u[k]), x.table) to u;
}

```

**Figure 13. Subroutine: Switch\_To\_S\_Node**

```

Action of y on receiving a InSysNotiMsg from x:
k = |csuf(y.ID, x.ID)|; N_y(k, x[k]).state = S;

```

**Figure 14. Action on receiving InSysNotiMsg**

**Lemma 5.1** Suppose node  $x$  joins a consistent network  $\langle V, \mathcal{N}(V) \rangle$ . Then, at time  $t_x^e$ ,  $\langle V \cup \{x\}, \mathcal{N}(V \cup \{x\}) \rangle$  is consistent.

**Lemma 5.2** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 2$ , join a consistent network  $\langle V, \mathcal{N}(V) \rangle$  sequentially. Then, at time  $t^e$ ,  $\langle V \cup W, \mathcal{N}(V \cup W) \rangle$  is consistent.

**Lemma 5.3** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 2$ , join a consistent network  $\langle V, \mathcal{N}(V) \rangle$  concurrently. If the joins are independent, then at time  $t^e$ ,  $\langle V \cup W, \mathcal{N}(V \cup W) \rangle$  is consistent.

**Lemma 5.4** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 2$ , join a consistent network  $\langle V, \mathcal{N}(V) \rangle$  concurrently. If the joins are dependent, then at time  $t^e$ ,  $\langle V \cup W, \mathcal{N}(V \cup W) \rangle$  is consistent.

To prove Lemma 5.4, first consider any two nodes in  $W$ ,  $x$  and  $y$ . If  $V_x^{Notify} = V_y^{Notify}$ , then  $x$  and  $y$  belong to the same C-set tree rooted at  $V_x^{Notify}$ , otherwise they belong to different C-set trees. We consider nodes in the same C-set tree first. We next present the definition of  $cset(V, W)$ , the C-set tree realized at time  $t^e$ . The definition is based on a snapshot of neighbor tables at time  $t^e$ .

**Definition 5.1** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 1$ , join a consistent network  $\langle V, \mathcal{N}(V) \rangle$ , and for any node  $x$ ,  $x \in W$ ,  $V_x^{Notify} = V_\omega$ ,  $|\omega| = k$ . Then the C-set tree realized at time  $t^e$ , denoted as  $cset(V, W)$ , is defined as follows:

- $V_\omega$  is the root of the tree.
- $C_{l_1 \cdot \omega}$  is a child of  $V_\omega$ , where  $C_{l_1 \cdot \omega} = \{x, x \in W_{l_1 \cdot \omega} \wedge (\exists u, u \in V_\omega \wedge N_u(k, l_1) = x)\}$ ,  $l_1 \in [b]$ .
- $C_{l_j \dots l_1 \cdot \omega}$  is a child of  $C_{l_{j-1} \dots l_1 \cdot \omega}$ , where  $C_{l_j \dots l_1 \cdot \omega} = \{x, x \in W_{l_j \dots l_1 \cdot \omega} \wedge (\exists u, u \in C_{l_{j-1} \dots l_1 \cdot \omega} \wedge N_u(k + j - 1, l_j) = x)\}$ ,  $2 \leq j < d - k$ ,  $l_1, \dots, l_j \in [b]$ .

Intuitively, in  $cset(V, W)$ ,  $C_{l_1 \cdot \omega}$  is the set of nodes in  $W_{l_1 \cdot \omega}$ , each of which is stored as a  $(k, l_1)$ -neighbor by at least one node in  $V_\omega$  by time  $t^e$ ;  $C_{l_2 l_1 \cdot \omega}$  is the set of nodes

in  $W_{l_2 l_1 \cdot \omega}$ , each of which is stored as a  $(k + 1, l_2)$ -neighbor by at least one node in  $C_{l_1 \cdot \omega}$  by time  $t^e$ , and so on. Next, we prove a few propositions about  $cset(V, W)$  and  $\mathcal{N}(V \cup W)$ , given that nodes in  $W$  belong to the same C-set tree. Propositions 5.1, 5.2 and 5.3 state that condition (1), (2) and (3), stated in Section 3.3, are satisfied at time  $t^e$ , respectively, while Proposition 5.4 concludes that by time  $t^e$ , nodes in the same C-set tree as well as nodes in  $V$  can reach each other. Then, Proposition 5.5 extends the result to nodes in different C-set trees. Our proofs of these propositions are based upon induction on C-set trees. Note that Propositions 5.1 to 5.4 make the following assumption:

**Assumption 5.1** (for Propositions 5.1 to 5.4)

A set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 2$ , join a consistent network  $\langle V, \mathcal{N}(V) \rangle$  concurrently and for any  $x$ ,  $x \in W$ ,  $V_x^{Notify} = V_\omega$ ,  $|\omega| = k$ .

**Proposition 5.1** If  $W_{l_j \dots l_1 \cdot \omega} \neq \emptyset$ ,  $1 \leq j \leq d - k$ ,  $l_1, \dots, l_j \in [b]$ , then  $C_{l_j \dots l_1 \cdot \omega} \neq \emptyset$ .

**Proposition 5.2** Let  $u$  be a node in  $V_\omega$ . If  $W_{l_1 \cdot \omega} \neq \emptyset$ ,  $l_1 \in [b]$ , then there exists a node  $x$ ,  $x \in W_{l_1 \cdot \omega}$ , such that  $N_u(k, l_1) = x$  by time  $t^e$ .

**Proposition 5.3** For any node  $x$ ,  $x \in W$ , if  $W_{l \cdot l_i \dots l_1 \cdot \omega} \neq \emptyset$ , where  $l \in [b]$  and  $l_i \dots l_1 \cdot \omega$  is a suffix of  $x.ID$ ,  $1 \leq i < d - k$ , then  $N_x(i + k, l) = y$  by time  $t^e$ ,  $y \in W_{l \cdot l_i \dots l_1 \cdot \omega}$ ; if  $W_{l \cdot \omega} \neq \emptyset$ ,  $l \in [b]$ , then  $N_x(k, l) = y$ ,  $y \in W_{l \cdot \omega}$ .

**Proposition 5.4** For any two nodes  $x$  and  $y$ ,  $x \in V \cup W$ ,  $y \in V \cup W$ ,  $\langle x \rightarrow y \rangle_d$  by time  $t^e$ .

**Proposition 5.5** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 2$ , join a consistent network  $\langle V, \mathcal{N}(V) \rangle$  concurrently. Let  $G(V_{\omega_1}) = \{x, x \in W, V_x^{Notify} = V_{\omega_1}\}$ ,  $G(V_{\omega_2}) = \{y, y \in W, V_y^{Notify} = V_{\omega_2}\}$ ,  $\omega_1 \neq \omega_2$ . Then by time  $t^e$ ,

- $\forall x, \forall y, x \in G(V_{\omega_1}), y \in G(V_{\omega_2}), \langle x \rightarrow y \rangle_d$ .

**Proof of Lemma 5.4:** First, separate nodes in  $W$  into groups  $\{G(V_{\omega_i}), 1 \leq i \leq h\}$ , where  $\omega_i \neq \omega_j$  if  $i \neq j$ , such that for any node  $x$  in  $W$ ,  $x \in G(V_{\omega_i})$  iff  $V_x^{Notify} = V_{\omega_i}$ ,  $1 \leq i \leq h$ . Then, by Propositions 5.4, 5.5 and Lemma 3.1, the lemma follows. ■

**Lemma 5.5** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 2$ , join a consistent network  $\langle V, \mathcal{N}(V) \rangle$  concurrently. Then at time  $t^e$ ,  $\langle V \cup W, \mathcal{N}(V \cup W) \rangle$  is consistent.

**Proof of Lemma 5.5:** First, separate nodes in  $W$  into groups, such that joins of nodes in the same group are dependent and joins of nodes in different groups are mutually independent, as follows (initially, let  $i = 1$  and put an arbitrary node  $x$ ,  $x \in W$ , in  $G_1$ ):

- For each node  $y$ ,  $y \in W - \bigcup_{j=1}^i G_j$ , if there exists a node  $x$ ,  $x \in G_i$ , such that  $(V_y^{Notify} \cap V_x^{Notify} \neq \emptyset)$  or  $(\exists u, u \in W - \bigcup_{j=1}^{i-1} G_j, (V_y^{Notify} \subset V_u^{Notify}) \wedge (V_x^{Notify} \subset V_u^{Notify}))$ , put  $y$  in  $G_i$ ;
- Pick any node  $x'$ ,  $x' \in W - \bigcup_{j=1}^i G_j$ , put  $x'$  in  $G_{i+1}$ , increment  $i$  and repeat these two steps until there is no node left.

Then, by Lemmas 5.4 and 5.3, the lemma holds. ■

**Theorem 1** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 1$ , join a consistent network  $\langle V, \mathcal{N}(V) \rangle$ . Then, at time  $t^e$ ,  $\langle V \cup W, \mathcal{N}(V \cup W) \rangle$  is consistent.

**Proof of Theorem 1:** According to their joining periods, nodes in  $W$  can be separated into several groups,  $\{G_i, 1 \leq i \leq l\}$ , such that nodes in the same group join concurrently and nodes in different groups join sequentially. Let the joining period of  $G_i$  be  $[t_{G_i}^b, t_{G_i}^e]$ ,  $1 \leq i \leq l$ , where  $t_{G_i}^b = \min(t_x^b, x \in G_i)$  and  $t_{G_i}^e = \max(t_x^e, x \in G_i)$ . We number the groups in such a way that  $t_{G_i}^e \leq t_{G_{i+1}}^b$ . Then, by Lemma 5.1 and Lemma 5.5, we conclude that at time  $t^e$ ,  $\langle V \cup W, \mathcal{N}(V \cup W) \rangle$  is consistent. ■

**Theorem 2** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 1$ , join a consistent network  $\langle V, \mathcal{N}(V) \rangle$ . Then, each node  $x$ ,  $x \in W$ , eventually becomes an S-node.

**Proof of Theorem 2:** First, consider a joining node,  $x$ , in status *copying*.  $x$  eventually changes status to *waiting* because it sends at most  $d$  *CpRstMsg* and each receiver of a *CpRstMsg* replies to  $x$  with no waiting. Second, consider a joining node,  $x$ , in status *waiting*. In this status,  $x$  sends *JoinWaitMsg* to at most  $d$  nodes. We next show that for each *JoinWaitMsg* it sends out,  $x$  eventually receives a reply. If the receiver of a *JoinWaitMsg*,  $y$ , is an S-node, then  $y$  replies with no waiting; if  $y$  is not yet an S-node, then it is a joining node in status *notifying* and will wait until it becomes an S-node before replying to  $x$ . Thus, to complete the proof, it suffices to show that any joining node in status *notifying* eventually becomes an S-node. Last, consider a joining node,  $z$ , in status *notifying*. There are two types of messages sent by  $z$  in this status, *JoinNotiMsg* and *SpeNotiMsg*.  $z$  only sends *JoinNotiMsg* to a subset of nodes in  $V \cup W$  that share the rightmost  $i$  digits with itself,  $i = z.noti\_level$ , and each receiver of a *JoinNotiMsg* replies to  $z$  with no waiting. Also,  $z$  only sends *SpeNotiMsg* to a subset of nodes in  $W$  that share the rightmost  $i + 1$  digits with it.<sup>8</sup> Each *SpeNotiMsg* is forwarded at most  $d$  times before a reply is sent to  $z$ , and each receiver of the message can reply to  $z$  or forward the message to another node with no waiting. Therefore,  $z$  eventually becomes an S-node. ■

## 5.2. Communication cost

Among the messages exchanged during a node's join, *CpRstMsg*, *JoinWaitMsg*, *JoinNotiMsg*, and their corresponding replies could be big in size since a copy of a neighbor table may be included, while messages of other types are small in size. We analyze the number of big messages in this section. The analyses for numbers of small messages are presented in [7].

For each message of type *CpRstMsg*, *JoinWaitMsg*, or *JoinNotiMsg*, there is one and only one corresponding reply.

<sup>8</sup>In simulations, we observed that *SpeNotiMsg* is rarely sent.

Hence, it is sufficient to analyze the number of messages for these three types. Theorem 3 presents an upper bound of the total number of *CpRstMsg* and *JoinWaitMsg* sent by a joining node,  $x$ . Next, let  $J$  be the number of *JoinNotiMsg* sent by  $x$ . The expectation of  $J$  when only  $x$  joins is given by Theorem 4, and an upper bound of the expectation of  $J$  when  $x$  joins concurrently with other nodes is given by Theorem 5. Proofs of the theorems are presented in [7].

**Theorem 3** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 1$ , join a consistent network  $\langle V, \mathcal{N}(V) \rangle$ . Then, for any  $x$ ,  $x \in W$ , the number of *CpRstMsg* and *JoinWaitMsg* sent by  $x$  is at most  $d + 1$ .

**Theorem 4** Suppose node  $x$  joins a consistent network  $\langle V, \mathcal{N}(V) \rangle$ ,  $|V| = n$ . Then, the expected number of *JoinNotiMsg* sent by  $x$  is  $\sum_{i=0}^{d-1} \frac{n}{b^i} P_i(n) - 1$ , where  $P_i(n)$  is  $\sum_{k=1}^{\min(n, B)} \frac{C(B, k) C(b^d - b^{d-i}, n-k)}{C(b^d - 1, n)}$  for  $1 \leq i < d - 1$ , where  $B = (b - 1)b^{d-1-i}$  and  $C(B, k)$  denotes number of  $k$ -combinations of  $B$  objects,  $P_0(n)$  is  $\frac{C(b^d - b^{d-1}, n)}{C(b^d - 1, n)}$ , and  $P_{d-1}(n)$  is  $1 - \sum_{j=0}^{d-2} P_j(n)$ .

**Theorem 5** Suppose a set of nodes,  $W = \{x_1, \dots, x_m\}$ ,  $m \geq 2$ , join a consistent network  $\langle V, \mathcal{N}(V) \rangle$ . Then for any node  $x$ ,  $x \in W$ , an upper bound of the expected number of *JoinNotiMsg* sent by  $x$  is  $\sum_{i=0}^{d-1} (\frac{n+m}{b^i}) P_i(n)$ , where  $n = |V|$  and  $P_i(n)$  is defined in Theorem 4.

Figure 15(a) plots the upper bound of  $E(J)$  when a set of nodes join concurrently, where  $n = |V|$  and  $m = |W|$ . We have implemented our join protocol in detail in an event-driven simulator. Figure 15(b) shows simulation results of the number of *JoinNotiMsg* sent by each joining node. We use the GT-ITM package [1] to generate network topologies. The topology used in Figure 15(b) has 8320 routers. There are two simulation setups. In one setup, 4096 nodes (end-hosts) are attached to the routers randomly, 3096 of which form a consistent network initially and the remaining 1000 nodes join concurrently. In the other setup, 8192 nodes are attached, and 1000 nodes join a consistent network formed by the other 7192 nodes. (In the simulations, all joins start at the same time.) For the simulations shown in Figure 15(b), average number of *JoinNotiMsg* sent by joining nodes are 6.117, 6.051, 5.026, and 5.399, respectively, while the upper bounds by Theorem 5 are 8.001, 8.001, 6.986, and 6.986, respectively. Also, results in Figure 15(b) indicate that the majority of joining nodes send a small number of *JoinNotiMsg*. Other simulation results show the same trend.

## 6. Discussions

### 6.1. Network initialization

The join protocol can be used for network initialization. To initialize a network with  $n$  nodes, put one node,  $x$ , in  $V$ , and construct  $x.table$  as follows:

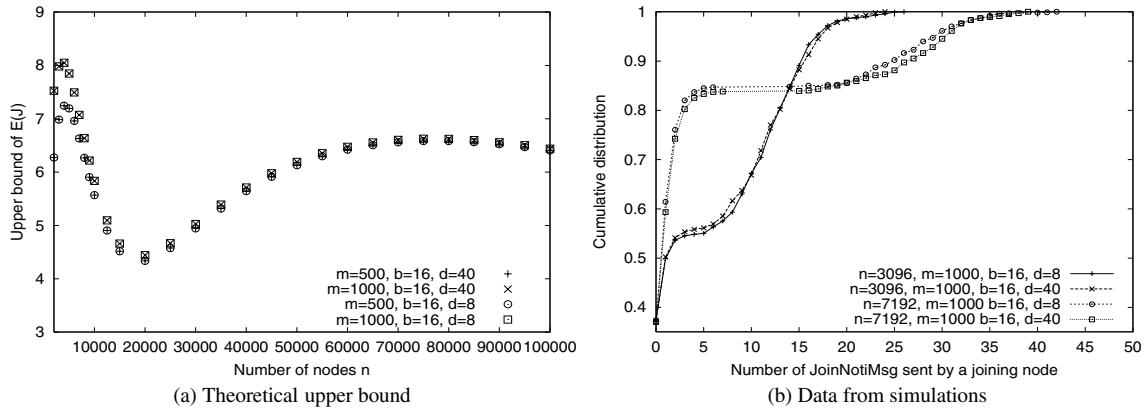


Figure 15. Number of JoinNotiMsg sent by a joining node

- $N_x(i, x[i]) = x, N_x(i, x[i]).state = S, i \in [d]$ .
- $N_x(i, j) = null, i \in [d], j \in [b]$  and  $j \neq x[i]$ .

Next, the other  $n - 1$  nodes join the network by executing the join protocol, each is given  $x$  to begin with. Eventually, a consistent network is constructed.

## 6.2. Message size reduction

In the join protocol, some types of messages need to include a copy of the sender's neighbor table. Several enhancements can be made to reduce the size of such a message:

- When node  $x$  sends a *JoinNotiMsg* to node  $y$ , it does not need to include its whole table in the message. Only including level- $i$ ,  $i = x.noti\_level$ , to level- $k$ ,  $k = |csuf(x.ID, y.ID)|$ , is enough.
- Moreover,  $x$  can include a *bit vector* in the *JoinNotiMsg* it sends to a node  $y$ , as suggested in [5]. Each bit corresponds to an entry in  $x.table$ , with '1' meaning that the entry is already filled and '0' meaning the opposite. Then, in its reply to  $x$ ,  $y$  only needs to include neighbors in level- $i$  entries that correspond to a '0' in the bit vector,  $0 \leq i < x.noti\_level$ , as well as all level- $i'$  neighbors,  $x.noti\_level \leq i' \leq d - 1$ .

## 7. Conclusions

For the hypercube routing scheme used in several proposed peer-to-peer systems [9, 13, 11, 6], we present a new join protocol that constructs neighbor tables for new nodes and updates neighbor tables in existing nodes. We present a rigorous proof that the join protocol produces consistent neighbor tables after an arbitrary number of concurrent joins. Furthermore, we present a conceptual foundation, C-set trees, for reasoning about consistency. We plan to use this conceptual foundation to design protocols for leaving, failure recovery, and neighbor table optimization. The expected communication cost of integrating a new node into the network is shown to be small by both theoretical analysis and simulations.

## Acknowledgment

The authors would like to thank Greg Plaxton for valuable discussions.

## References

- [1] K. Calvert, M. Doar, and E. W. Zegura. Modeling Internet Topology. *IEEE Communications Magazine*, June 1997.
- [2] M. Castro, P. Druschel, Y. C. Hu, and A. Rowstron. Exploiting network proximity in peer-to-peer overlay networks. In *Proc. of International Workshop on Future Directions in Distributed Computing*, 2002.
- [3] Freenet. <http://freenetproject.org>.
- [4] Gnutella. <http://www.gnutella.com>.
- [5] K. Hildrum, J. D. Kubiawicz, S. Rao, and B. Y. Zhao. Distributed object location in a dynamic network. In *Proc. of ACM Symposium on Parallel Algorithms and Architectures*, 2002.
- [6] X. Li and C. G. Plaxton. On name resolution in peer-to-peer networks. In *Proc. of the 2nd Workshop on Principles of Mobile Computing*, 2002.
- [7] H. Liu and S. S. Lam. Neighbor table construction and update in a dynamic peer-to-peer network. Technical Report TR-02-46, Dept. of CS, Univ. of Texas at Austin, <http://www.cs.utexas.edu/users/lam/NRL/>, Sept. 2002.
- [8] Napster. <http://www.napster.com/>.
- [9] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. of ACM Symposium on Parallel Algorithms and Architectures*, 1997.
- [10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of ACM SIGCOMM*, 2001.
- [11] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.
- [12] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of ACM SIGCOMM*, 2001.
- [13] B. Y. Zhao, J. D. Kubiawicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Aug. 2001.